



IFW

RECEIVED

MAR 15 2007

OFFICE OF PETITIONS

From:
Yefim (Jeff) Zhuk
11191 East Ida Place, Englewood, Colorado 80111

To: USPTO, Petition office

Petition to withdraw holding of abandonment under 37 CFR1.181

To whom it may concern:

In the first week of August, 2006 I received a Non-Final Rejection on my patent Application Number: 10/709460, Knowledge Driven Architecture.

I wrote a response and on October 23, 10:50am (N.Y.) faxed it to USPTO office, tel. 5712728300.

I asked my son Ben Zhuk to use his MaxEmail Fax service to fax the response to the USPTO office. He faxed the claims and the abstract page (total 5 pages) that I updated based on reviewer objections and recommendations.

My son sent me the receipt and I was sure that my response was delivered to USPTO on time. Here is the receipt from the Fax service enclosed.

In the first week of March, 2007, I received a phone call from my application reviewer, who let me know that no response was received during 6 month and the application must be abandoned. I immediately emailed to him my response file but he explained to me that the right process is to abandon application first and then write this petition to revive the application.

I enclose my response with the Abstract, Claims, and the body of the patent.

I would like to ask for your generous permission to revive the application review process.

Thank you so much for your time and considerations.

Yefim "Jeff" Zhuk
720-299-4701

Yefim Zhuk

3/8/07



Application Number: 10/709460
EFS ID: 60540
Timestamp: 2004-05-06 21:58:17 EDT
The patent was Electronically filed on May 6, 2004
Provisional Patent #60532384 12/26/03.

Trademark Notice of Publication Under 12(a) 04/28/04
Serial No.: 78/289,572
Mark: Knowledge Driven Architecture
Publication Date: 05/18/04
Applicant: Internet Technology School, Inc.

Knowledge Driven Architecture

Abstract

A method and system that transforms application scenarios written with ontology expressions into software applications. The method and system in one embodiment speeds up software system development and provides a greater flexibility to system behavior. Developers or subject matter experts can omit multiple steps of translating application requirements into traditional programs, and instead directly drive applications with application scenarios written in "almost natural" language of ontology expressions and stored in the knowledgebase.

The system further comprises of an application scenario player, a service connector, presentation components, and underlying knowledgebase and application services.

This description is not intended to be a complete description of, or limit the scope of, the invention. Other features, aspects and objects of the invention can be obtained from a review of the patent, the figures and the claims.



Untitled

-----Original Message-----

From: MaxEmail Send [mailto:SendAdmin@maxemail.com]
Sent: Monday, October 23, 2006 10:50 AM
To: bzhuk1@yahoo.com
Subject: MaxEmail Send Delivery Report Job 21406257

Maxemail Send Job Confirmation For Job ID 21406257

Job Information

Maxemail Job ID : 21406257
Number of Pages : 0005
Recipient Count : 1
Report Time Zone: America/New_York (GMT-0400)

Recipient Delivery Summary

Delivered : 0
Errorred : 1

Rec Fax Number	Pgs	Duration	Calls	Status
0000 5712728300	0005	00:00:00	1	Delivered

End Of Report



Background of the Invention. There is a gap in the current development process between the initial business input provided by business experts and the final implementation. The current process requires multiple transformations from user requirements into low-level programming functions and data tables. Multiple technology teams work hard to create multiple filter-layers to fill this gap and break the beautiful shapes of reality into small and simplistic pieces, transforming business rules into Boolean logics.

This process hasn't changed much during the last twenty years.

Some time ago, we thought of the C-language as a language for application development, while Assembly was the language for system development. Since the Assembly language was pushed down to the system level, we use languages like C, C++, C#, VB, and Java to code business algorithms in applications. This development paradigm has been in place for about 20 years.

However, the current shift to service-oriented architectures and recent advances in knowledge technologies can allow us to cleanly separate generic services from application-level business rules and specific requirements of user-program or program-program interfaces.

Using a knowledgebase, we can represent requirements as business rules described in "almost natural" language. We can finally shift our focus from ironing out all possible business cases in our design and code to creating flexible application mechanisms that allow us to change and introduce new business rules on-the-fly.

We can also improve the pattern recognition process. Imagine that a set of rules which defines a recognition process, for example in a grammar file, is enhanced with the ability to access a knowledge engine. This would promise more intelligent recognition, based not only on the multiple choices prepared in the file, but also on existing facts and rules of the available world of knowledge, associations between related topics, etc.

Software applications often represent a new combination of existing software components. These components often need to be polished or changed to fit into a new application mosaic. Service components are glued and compiled together with specific business rules, expressed as programming algorithms that distinguish the application. Creating an application is a project, often a big project, for multiple teams; the initial team of business experts that know "what should be done" is just the tip of the iceberg. This invention can improve the development process by decreasing the transformation steps required, and thus stop the business requirements from getting watered down and distorted during the process.

SUMMARY OF THE INVENTION

The invention provides a better separation between generic service components and specific rules and scenarios that characterize the application. Services in a knowledge-driven architecture are integration-ready components presented at the service layer locally or distributed over the network. Business rules and scenarios that represent a very light application layer can be created and changed at run-time by business experts, who would have their chance to influence application behavior, and to say not only "what should be done," but also "how".

The main component of the application is the knowledgebase, which stores application requirements as application scenarios and business rules. The Application Scenario Player and the Service Connector transform application scenarios into interactions with the knowledgebase, presentation components, and underlying application services.

Knowledge-driven architecture includes elements and mechanisms providing growing awareness about knowledge and services existing on distributed knowledge systems built with this architecture.

These mechanisms and collaborative mechanisms described in the Distributed Active Knowledge and Process Base (see Yefim (Jeff) Zhuk, Patent No. US 7032006 B2, <http://uspto.gov>) allow separate systems to negotiate multiple forms of collaboration: to share or trade knowledge and services with sufficiently flexible levels of security.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention can be more readily understood in conjunction with the accompanying drawings, in which:

Fig. 1 is a diagram that shows the major blocks of knowledge-driven architecture.

Fig. 2 shows details of the *Presenter* and *knowledgebase* components

Fig. 3 is a collaboration diagram that describes the interaction between the components

Fig. 4 illustrates details of the *Service Connector* component.

Fig. 5 discloses details of the *Scenario Player*.

Description

Turning to FIG. 1, the present invention consists of the knowledgebase 100 as the main component that collaborates with the application scenario player 200 and the service connector 300. The application scenario player 200 interprets application scenarios and interacts with the service connector 300 and presentation components 400. The service connector 300 provides access to the knowledgebase 100 and service components 500. This is the Functional View (or the Component View) of the architecture. Application requirements are directly represented by the business rules stored in the knowledgebase. A flow of application information as well as interactions between users and the program or/and between partner programs are described by application scenarios written with an XML based application scenario language (ASL).

We can review the diagram from the position of the Model-View-Controller Design Pattern.

From this point of view, the knowledgebase with business rules and scenarios, together with application service components, represent the Model. The presentation components represent the View. The Scenario Player 200 and the Service Connector 300 represent the Controller.

The major component of the Model is the knowledgebase or Knowledge Engine (KE) 100. Business rules are captured in the knowledgebase with an "almost natural" ontology language like CycL. A user or program agent 600 can also transmit an application scenario to the controller. Application scenarios are written in the Application Scenario Language. The Application Scenario Language allows developers or business experts to describe application flow as a set of scenario acts with conditional service invocations. The scenario acts can describe interactions between users and programs or between partner programs engaged in a common business transaction, and include calls to knowledgebase services for application business rules and related data.

Traditional services, which are written with current programming languages like C# or Java and compiled into binary service components, can also capture some business rules and algorithms to support the Model. Services are designed as integration-ready components with separate APIs required by the Service Connector 300. The Service Connector 300 transforms service requests from application scenario acts into direct calls to service components 500.

Each application scenario is a set of scenario acts. Each act is a very lightweight XML description that can invoke some services, exercise conditions or business rules, and include rules for the interpretation of expected user or program agent responses.

There is a traditional View block of the MVC design pattern. The View delivers information in selected video, sound, or electronic formats; people as well as partner programs are its target audience. In the future, we will refer to presentation components or the View as the Presenter 400.

The Scenario Player 200 is connected via the Service Connector 300 to the knowledgebase 100 and other services 500 that represent a business model, and actively uses the knowledgebase 100 in the process of application scenario interpretation.

The input information for the Scenario Player 200 can be data entered by a user via voice or any other method, an XML service request from the network, or an act of an application scenario. The Scenario Player 200 interacts with the Service Connector 300 that provides access to the knowledgebase 100 and traditional services 500, as well as to the Presenter 400, which transforms resulting data into the proper format. The format depends on two factors. A specific implementation of the Presenter 400 that gears towards specific video, audio, or electronic formats is a fixed factor. An application scenario can include some presentation definition-rules that provide extra flexibility for the presentation layer.

The Scenario Player 200 is also responsible for the interpretation of service or knowledgebase responses. Interpreted responses are directed to the Presenter 400, which performs the final transformational steps and delivers data to the target audience in the selected presentation format.

The Presenter 400 can include special engines, like speech, handwriting, or image recognition, which might target a specific type of user input.

Scenarios can describe sequences of expected events related to multiple agents 600, and provide rules on handling these events. These scenarios will map each expected event to its observer object, or a set of observers that have interest in the events and handle them with the proper services 500.

The functionality described above provides for great flexibility, but can suffer in performance. The Optimizer 700 takes a snapshot of existing rules and scenarios and translates them into source, for example in Java or C#. This source can later be compiled into binary code to iron the current status of application rules into a regular application that lacks flexibility but provides better performance.

Fig.2 shows details of the Presenter 400 and knowledgebase 100 components.

The Presenter 400 can include the Communicator 420, the Performer 440, and the Formatter 460 components.

The Formatter 460 prepares data for audio or video interaction or for communication to other programs. HTML is an example of such formatting. The Performer 440 component uses formatted data for actual presentation via voice or screen. The Performer 440 can be implemented in multiple ways for different client device types. For example, the Performer 440 can display HTML data via the rich graphical interface of a thick client device or workstation. The Communicator 420 is responsible for formatted data communications via peer-to-peer distributed networks or other protocols.

Knowledge and service elements can be distributed over the network where they can promote their capacities and can be accessed via Communicator 420 using collaborative mechanisms described in the Distributed Active Knowledge and Process Base (see Yefim (Jeff) Zhuk, Patent No. US 7032006 B2, <http://uspto.gov>).

The knowledgebase 100 component includes the knowledge service component 120 and the knowledge engine 140.

The *KnowledgeService* 120 component serves as the adapter to the knowledge engine 140, and adapts the knowledge engine interface to the interface required by the Service Connector 200.

The collaboration diagram in Fig. 3 describes interaction between the components.

The diagram simplifies the activity of the application to 8 major steps.

1. The *ScenarioPlayer* object receives an XML instruction from the network, as an act of a current scenario, or a user's input related to the scenario.
2. The *ScenarioPlayer* interprets this input and translates it into a service request directed to the *ServiceConnector* (the most common action).
3. The *ServiceConnector* object uses its *act()* method to connect to or obtain (load at run-time) a necessary service object that will perform the requested operation/method.
4. The service object invokes the proper method with the necessary parameters, and delivers results back to the caller.
5. The *ScenarioPlayer* gets the results of the service, and passes them further to the *Formatter* object.
6. The *Formatter* implementation translates results into a presentation format and produces XML scenarios related to the expected user interaction.
7. The *ScenarioPlayer* interprets the results for the *Performer* object if the operation was a success. If the operation failed (for example, the knowledgebase query returns a "not found" string), the *ScenarioPlayer* can use the Communicator peer (if present) to outsource this operation to a network of knowledge peers.
8. The *Performer* object presents results on a screen or/and in a voice format. The alternative for this step is to communicate data to other peers (for example, if the local peer failed, another peer on the network may be able to resolve the request) or to partner programs.

The eight steps of the cycle are now completed. The newly retrieved scenario not only includes data shown on a screen or pronounced via voice. There is an invisible part related to expected agent (user or program) interactions. This portion of the information is formatted as a set of scenario acts, where each act is mapped to a specific user choice or agent event. The program is again ready to start from point one of the collaboration diagram to react to a user's input or play the next act of a scenario.

Fig.4 illustrates the details of the *ServiceConnector* component.

Each component in the figure represents a block of software responsible for the proper functionality of a component.

The Actor 310 block is able to play multiple object roles. It takes service and action names as parameters, and invokes the requested method on the requested object type.

The Actor 310 receives the name of the service and the name of the operation to perform with the required parameters. The Actor 310 looks into the Object Registry 330 to check if there is an existing object of the requested service class. If there is no such service object yet, the Actor 310 works with the Object Retriever 320 to load the requested service class and instantiate the object

of this class at run-time. The Actor 310 then registers (stores) the retrieved object in the Object Registry 330.

Multiple objects of the same service class are associated with object names. In this case, the Service Connector 200 receives an object name as an additional argument to service and action names that identify the service class and the method names. The registered object keeps its state during its lifetime, which can include many service invocations.

The next step is to use Method Retrieval 340 to retrieve the proper method, which will perform the requested service operation. The Method Retrieval 340 selects one of the methods of the selected service object based on the provided method parameters. The Actor 310 then uses the Method Performer 350 to perform the operation.

The Service Connector can be implemented in Java, C#, or other languages that allow systems to load service objects at run-time based on their names. The Method Retrieval mechanism consists of three steps/trials.

1. Use the method name and parameter types to find an exact match. For example, a Java implementation would use the mechanism offered by the *Class.getMethod()* method.
2. Unfortunately the exact match rarely happens in real programs. Parameter types are often subclasses of required argument classes.
The second step is to check for method compatibility instead of the exact match. For example, a Java implementation would use the mechanism offered by the *Class.isAssignableFrom()* method.
3. It is possible that both trials will fail because of implementation issues. For example, Personal Digital Assistants (PDA) or wireless phones do not have sufficient library support for such sensitive reflection mechanisms. In this case, the third attempt will invoke a specific method of the service object that serves as the dispatcher for other methods of this service class. For example, it may call a method named "dispatch()" of that class according to the initial interface/agreement between the Service Connector and service classes. The "dispatch()" method takes the name of a service operation (a method name) and the array of objects. These objects will be cast into specific types inside the "dispatch()" method according to the requirements of the specified service method.

The *ScenarioPlayer* 200 component is responsible for handling agent events. The *ScenarioPlayer* 200 also provides interpretation and performance of application scenarios, which contain rules for possible events and related handling procedures.

Requested services can be implemented as service components 500 or as knowledgebase rules.

Fig.5 discloses details of the Scenario Player 200.

The Scenario Player 200 consists of two major parts: the Interpreter and the Act Player presented in Fig.5 with blocks 210-240 and 250-290 respectively.

The Input Type Checker 210 checks current input and, depending on its type, submits the input to one of the interpreters.

Knowledge-driven architecture includes elements and mechanisms providing growing awareness about knowledge and services existing on distributed knowledge systems built with this architecture. These mechanisms include but not limited by the Service Analysis 215 component and learning scenarios.

The Success Analysis 215 component provides a history of success, history of interpretation failures, and, in the case of interpretation failure, invokes one of the learning scenarios that prompt an agent (a user or a program) to re-define the input or provide more details for better interpretation. If the learning scenario cannot be executed at that time (it was canceled, etc.), the scenario will be placed in the queue of scenarios with un-answered questions to be played later and to resolve unsuccessful interpretations.

Each service component has usage and value properties (see Yefim (Jeff) Zhuk, Patent No. US 7032006 B2, <http://uspto.gov>). The Success Analysis 215 component re-fines these properties after each service request.

The set of interpreters includes but is not limited to the Scenario Act Interpreter 220, the Prompt Response Interpreter 230, and the New Agent Request Interpreter 240.

All interpreters transform original input into scenario player APIs or service component APIs. Interpreters are connected to the Presenter 400 and can use the Formatter 460 and the Performer 440 services. For example, the input line can instruct the system to present information via a specified video or audio format. It is also possible that no current interpretation will be found. In this case, a default learning scenario would be played, prompting an agent (a user or a program) to re-define the input or provide more details. Default scenarios can be replaced at run-time with enhanced ones.

Interpretation rules are stored in the knowledgebase 100. Interpreters interact with the Query Performer 280 to access the knowledgebase 100 via the Knowledge Service 120. When existing rules fail, the learning scenario (default or not) is given the questionable unresolved input, and is called upon to retrieve new definitions or more details from an agent (a user or a program). Upon successful execution, the scenario ends up with one or more acts that re-define existing rules or/and add more rules to the knowledgebase. If the learning scenario cannot be successfully executed at this time, the scenario will be stored in the Queue of Scenarios 245, and will be tried again later.

This provides for great flexibility, which is a welcomed feature for most business applications as well as educational systems.

The Player part of the Scenario Player 200 consists of the Queue of Scenarios 245, the Scenario Modifier 250, the Current Scenario 255, the Next Act Retrieval 260, the Translator 265, the Alias Retrieval 270, the Condition Checker 275, the Query Performer 280, and the Service Performer 290.

The Queue of Scenarios 245 stores the current scenario when it cannot be executed at the present time, but needs to be executed later on. For example, if a user cannot provide answers to a learning scenario at this moment, but can do so afterward.

The Scenario Modifier 250 receives results from every step of playing the scenario act. If the current act resolves the value of any variable in the scenario, the Scenario Modifier 250 replaces this variable with its value in the Current Scenario 255. For example, the scenario can include the variable PEER-GROUP-NAME. Any act of the scenario that resolves this variable will pass the variable name and its value to the Scenario Modifier 250, and the Scenario Modifier 250 will replace this variable in the Current Scenario 255 with the value of the variable.

The Current Scenario 255 is loaded from the knowledgebase 100 or from the Queue of Scenarios 245, and can be updated with run-time values by the Scenario Modifier 250. Any current scenario consists of scenario acts: simple XML elements/instructions. Each instruction can be a prompt to an agent (a user or a program) or a service request, including internal and external services.

The Next Act Retrieval 260 retrieves one act of a scenario at a time. This is usually the next act according to the sequence of acts stored in the scenario. The sequence of acts can be changed with conditional statements.

Blocks 265-290 in Fig.5 can be considered as examples of internal services that are often used in the process of interpreting instructions or as responses to a prompt.

The Translation 260 takes a set of arguments that describe the type of the requested translation. The **translate** element invokes the Translation 260 block. Here is an example of using the **translate** element:

```
<prompt variable="PERSON-NAME" action="prompt"
  service="com.its.connector.ScenarioPlayer"
  msg="Please provide your name (First Last)"
  translate=
    "concatenate(REPLACE-WITH-INPUT)^startWithUpperCase(REPLACE-WITH-INPUT)"
/>
```

The system will find the *concatenate* and *startWithUpperCase* methods in the available translator service components or in the knowledgebase where the action is stored as an executable rule. The action will concatenate a user's input and make sure that each word begins with the upper case. For example, if the input was "jeff zhuk" the first instruction will produce "jeffZhuk" and the second instruction will make it "JeffZhuk".

The Alias Retriever 270 checks for in-line aliases that can be used as the **"aliases"** element in the current scenario act. For example, the following element would instruct the Alias Retrieval 270 to interpret prompt responses like "Y", "OK", or "Sure" as "Yes".

```
aliases="Yes|y|ok|sure"
```

The Alias Retriever 270 can be also invoked when an instruction includes the **"inAliases"** element that directs the Alias Retrieval 270 to look in the knowledgebase for a set of related aliases as provided in the example below.

```
<prompt variable="TRAINING-COURSE" perform="prompt"
  msg="What subject do you want to learn?" inAliases="JavaTrainingCourses" />
```

The knowledgebase can contain a set of rules/aliases related to course names. For example, entered keywords like "enterprise" or "server" will result in the name "J2EE", while keywords like "PDA" or "wireless" will result in the course name "J2ME".

A found alias, for example, "J2EE", will be passed to the Scenario Modifier 250 to replace the current variable name "TRAINING-COURSE" with its alias value. Otherwise, the variable name will be replaced with the original response.

The Condition Checker 275 is invoked by the **"condition"** element in a scenario act. The **"condition"** element is followed by a specified condition, one from the list of conditions, (like "exists", "!exists", "equals", etc., see the list of conditions in the application scenario language) and required arguments.

A conditional statement is usually followed by an action to perform. If a condition is not met (returns false) the action will not be performed, and the next scenario step will be played instead.

For example, a conditional instruction can check if a requested training course exists.

```
<if condition="!exists" pattern="TRAINING-COURSE"
    lastMsg="The course is not found. Enter more keywords." />
```

Conditional instructions, like "includes", "equals", etc. require two arguments: a pattern and a source.

```
<if condition="includes" pattern="TRAINING-COURSE" source="XML"
    perform="playScenario(XMLTechnologies)" />
```

The conditional instruction in the example above will check if a selected training course includes the "XML" keyword. If the condition is true the system will start playing the "XMLTechnology" scenario. Otherwise the next sequential act of the current scenario will be played instead.

The Query Performer 280 is invoked by the **query** element when it is present in a scenario.

```
<act queryResult="PASSWORD-QUESTION=PASSWORD-ANSWER"
    query="passwordOf(USER-NAME)" />
```

The query instruction provided in the example above checks in the knowledgebase for a user's record and delivers one of the password questions with its answer. In the current example, knowledgebase records include more than one way to check a user's identity. There may be different password questions as well as password answers, and the knowledgebase would select one or more of them based on some rules established by your requirements.

For example, the question can be as simple as "Password?", or more complicated like "What is your mother's maiden name?", etc. The question is retrieved along with the answer, and both are assigned to proper variable names.

In the case of a successful query, the retrieved value will be passed to the Scenario Modifier 250 to replace the variable name in the current scenario.

The Service Performer 290 is invoked by the **action** or **perform** elements at the very end of the execution of an act of a scenario regardless of the order in which the elements of the act were written. Translations, alias retrievals, conditions, and queries, if any, are done before the Service Performer invocation.

The Service Performer 290 transforms the *action* or *perform* elements into a set of arguments including a service name, an optional object name (if multiple objects of the same service class are to be used), a service operation name, and a set of parameters. It then passes these arguments to the Service Connector 200, which will access the proper service object and execute the proper service method.

Application Scenario Language

Application Scenario Language (ASL) is an XML-based language that describes application business flow in small scenarios. The scenario language constructs can be changed, improved, and extended; they are here in this form to illustrate the invention.

Scenarios consist of XML elements: scenario steps or acts.
Every act of a scenario is a prompt, a condition, or an execution step.

Prompt the user or a partner program for an answer

The prompt might have additional arguments, specific rules for input interpretation, and conditional actions.

Here is an extract from the *addKnowledge.xml* scenario, which allows someone to introduce a new object to the knowledgebase.

```
<prompt variable="NEW-OBJECT"
  service="com.its.connector.ScenarioPlayer"
  action="prompt"
  noinput="reprompt(Your input is needed)"
  translate="concatenate"
  msg="Please provide a name for your new topic." />

<if condition="!exists" perform="doNextStep(acceptNewObject)" />

<act name="reject" action="query" constant="NEW-OBJECT"
  lastMsg="NEW-OBJECT is not new." />

<act name="acceptNewObject" service="com.its.connector.KnowledgeService"
  action="createNewPermanent" constant="NEW-OBJECT" />
```

The prompt element of the scenario will invoke the prompt mechanism (method) of the *ScenarioPlayer 200*. The *prompt()* method of the *ScenarioPlayer* class works with the *Presenter 400* component to deliver a prompt message. The method shifts the *ScenarioPlayer* into the interpretation state. It will interpret the user's response and assign the **variable** provided with the **prompt** parameters to the value of the interpretation result.

The prompt element of the XML scenario specifies the service-class name (*com.its.connector.ScenarioPlayer*) and the action-method name (*prompt*), and sets the prompt variable (*NEW-OBJECT*) to store the user's input. One of the most important arguments of the prompt element is the prompt message delivered to the target audience.

The *noinput* and *translate* elements are optional interpretation parameters. The *noinput* element directs the program to re-prompt a user if the user just pressed the ENTER key.

The *translate* element instructs the program to concatenate multi-word input into a single word that can better serve as a unique reference.

A more complete example of the Application Scenario Language is provided in the attached Appendix I - APS.doc

By providing application requirements via business rules with "almost natural" predicate logics, we can take a short-cut past several steps of the traditional development process, where requirements are boiled down to traditional Boolean-logics based programs. The main reason is the difference between programming languages and ontology languages, which can be used to describe business rules. Ontology languages are closer to our natural language, and can express an unlimited number of relationships that can be provided with predicates, while programming languages are extremely limited with their syntax and especially with their set of relationships. The difference between predicate logics and Boolean logics is tremendous. In a way, traditional development translates natural language requirements into Boolean logics. Predicate logics require almost no translation.

In the example below, we use the Cyc language (Cycl), by Cyc Corporation, to describe several rules of an educational system. Let us say that the application allows students and instructors to collaborate in a group with multiple roles. A three-dimensional matrix of roles, related access types, and privileges describes access to documents and services (see Yefim (Jeff) Zhuk, Distributed active knowledge and process base, Patent No US 7032006 B2, <http://uspto.gov>).

A very powerful and simple Cyc Language constant helps create unlimited hierarchies. The formula below means that every instance of the first collection, *GroupMember*, is also an instance of the second collection, *SystemUser*.

```
(genls GroupMember SystemUser)
```

In other words, *SystemUser* is a generalization of *GroupMember*.

The *genls* predicate expresses the idea that one collection is subsumed by another.

In the example below we can define a new function that will return the group role of a member.

Here is an example of a definition for the function *MemberRoleFn*:

```
(arity MemberRoleFn 2)
(arg1Isa MemberRoleFn User)
(arg2Isa MemberRoleFn Group)
(resultIsa MemberRoleFn GroupRole)
```

We read this function definition as: the *MemberRoleFn* function has 2 parameters: user and group. The function returns the specific group role that the user plays in the specified group.

The following example establishes a rule with the *implies* keyword.

```
(implies
  (and
    (hasMembershipIn ?USER ?GROUP)
    (hasRole ?USER ?GROUP Admin)
    (hasPrivilege ?USER ?GROUP ChangeMemberRoles)))
```

This rule says that if a user has membership in a group and the user has the role of an administrator in this group – this user has the privilege to change member roles in this group.

“*implies*”, “*and*”, as well as “*or*”, and “*not*” are important Cyc’s logical connectives.

The benefits of using an ontology language versus a programming language become even more impressive when application rules are more complex and have a tendency to change frequently.

Application scenario language allows us to describe business information flow and user or program interactions. Application scenarios complement business rules, providing direct access to application services. Application scenario language constructs describe service and operation names, define data flow conditions and provide interpretation rules for successful business operations. Application scenarios as well as business rules that comprise the application layer can be directly created by business experts, while application services will be created by programmers and can be considered as part of the system layer.

Here is an example of the scenario for an educational system. The scenario allows the system to accept a new object and add this object to the knowledgebase, and provides integration of the new object with existing knowledge.

```
<prompt variable="NEW-OBJECT"
  perform="prompt"
  noinput="reprompt(Your input is needed)"
  translate="concatenate"
  msg="Please provide a name for your new topic." />
```

```
<if condition="!exists" perform="doNextStep(acceptNewObject)" />
```

```
<act action="query" constant="NEW-OBJECT"  
lastMsg="NEW-OBJECT is not new." />
```

```
<act name="acceptNewObject" service="com.its.connector.KnowledgeService"  
action="createNewPermanent" constant="NEW-OBJECT" />
```

```
<prompt variable="EXISTING-COLLECTION" perform="prompt"  
msg="Enter an existing topic name that can serve as a parent to your new topic." />
```

```
<if condition="!exists" pattern=" EXISTING-COLLECTION"  
perform="reprompt(EXISTING-COLLECTION is not found in the KB.) " />
```

```
<act query="(isa EXISTING-COLLECTION ?X)  
queryResult="COLLECTION-QUERY-RESULT" />
```

```
<if condition="!includes"  
pattern="Collection"  
perform="reprompt(EXISTING-COLLECTION is not a Collection)" />
```

```
<act lastMsg="NEW-OBJECT is integrated in the knowledgebase" />
```

The example above prompts a user to provide an existing collection name that would relate a new subject to existing data. The following statements check if the user's input is worth trusting.

Does this parent name really exist in the knowledgebase, or just in the user's imagination? Does this name meet the requirement to be a collection? (Not every existing object can be a parent. Only Collection type objects can.)

This short scenario written by business expert can be easily extended or complemented by other scenarios without a programmer's participation.

An example of constructs of Application Scenario Language is provided in the attached document named "Application Scenario Language".

Claims

1. A method of transforming application scenarios written with ontology expressions into software applications, comprising: translating business rules, captured as ontology expressions in a knowledgebase, into an application scenario; transforming the actions in said application scenario into interactions with at least one of: a knowledgebase, one or more presentation components, and underlying services.

2. The method of claim 1, further comprising at least one of: separation of software in the application business layer, which describes business rules as ontology expressions and service calls, and the application service layer, which describes services; creation and modification of business rules and scenarios that comprise the application business layer at run-time; analysis of successful scenario execution including at least one of: history of successes, history of interpretation failures, learning scenarios that prompt an agent (a user or a program) to re-define the input or to provide more details for better interpretation; a queue of scenarios with unanswered questions to resolve unsuccessful interpretations.

3. The method of claim 2, further comprising: awareness of and interaction with knowledgebase and service elements existing on distributed network systems built with this architecture; at least one method enabling collaboration and sharing of knowledge and service resources over distributed networks.

4. The method of claim 2, further comprising: translation of existing application scenarios into source code for traditional fixed-form applications with better performance but less flexibility.

5. A system for storing and executing application scenarios, comprising: a knowledgebase containing knowledge facts and business rules captured as ontological expressions, as well as sets of said business rules called application scenarios; a Scenario Player that is able to transform actions captured in application scenarios into interactions with at least one of a knowledgebase, presentation components, and underlying services.

6. The system of claim 5, wherein application scenarios are written with the Application Scenario Language, which describes application flow as a set of scenarios that define interactions between system components and agents, where said agents can be users, human beings, or programs.

7. The system of claim 6, wherein application scenarios consist of scenario actions that can define any of:

- (i) interactions between system components and agents
- (ii) prompt messages to agents, expected agent responses, if any, and rules for interpretation of agent responses
- (iii) invocations of knowledgebase services, like queries, assertions, etc. using service names and optional arguments
- (iv) invocations of application services using service and operation names and optional arguments
- (v) variables that are replaced with their values at run-time
- (vi) conditions based on run-time variable values and knowledgebase queries
- (vii) the order of execution of scenarios and scenario acts
- (viii) aliases, or multiple ways of expressing the same meaning
- (ix) translation policies related to input
- (x) possible agent events and event handling rules
- (xi) instructions for presentation components

8. The system of claim 5, further comprising a Presenter component that includes one or more of:
(i) Formatter that prepares data for audio or video presentation or for communication with other programs

- (ii) Communicator that provides formatted data communications via peer-to-peer distributed networks or any other protocols to provide collaborative access to knowledge and services existing on distributed network systems built with this architecture
- (iii) Performer that uses formatted data for actual presentation via voice or screen
- (iv) Special engines such as speech, handwriting, or image recognition, which can target specific types of user input.

9. The system of claim 5, further comprising a Service Connector component, which includes:

- (i) Object Retrieval that is able to find an existing service or load the requested service definition and instantiate the service object at run-time
- (ii) Object Registry that associates service objects with service and object names, stores service objects, and makes them reusable
- (iii) Method Retrieval that retrieves the proper service method belonging to a selected service object based on the provided method arguments
- (iv) Method Performer that performs the requested service operation on the selected service object

10. The system of claim 9, further comprising a Knowledge Service component, which serves as an adapter to the knowledgebase, adapting the knowledge engine interface to the interface required by the Service Connector.

11. The system of claim 5, further comprising an Optimizer component that takes a snapshot of existing rules and scenarios and translates them into source code in a language such as Java or C#, which can later be compiled into binary code to fix the current application rules into a regular application that lacks flexibility but provides better performance.

12. The system of claim 5, wherein the Application Scenario Player contains but is not limited to at least one of

- (i) Input Type Checker that checks current input and, depending on its type, submits the input to one of several interpreters
- (ii) One or more interpreters that translate acts of scenarios or any other input into a direct action by one of the system components, based on scenario and knowledgebase rules
- (iii) Queue of Scenarios that stores the current scenario when it cannot be executed at the current time, but needs to be executed later.
- (iv) Success Analysis component that can store and retrieve a history of interpretation successes and failures and, in the case of interpretation failure, can invoke one of several learning scenarios that prompt an agent (a user or a program) to re-define the input or to provide more details for a better interpretation.

13. The system of claim 5, wherein knowledge facts, service, and scenario components are endowed with usage and value properties.

14. The method of claim 5, further comprising: creation and modification knowledge facts, services, and composite service scenarios

15. The method of claim 5, further comprising: initial definition of the usage and value properties of newly created facts, services, and scenarios and ability to change these values at run time

16. The system of claim 15, wherein a Success Analysis component maintains and consistently refines a list of previously used services with their APIs, keywords, descriptions, and related scenarios in the knowledgebase, and re-evaluates the usage and value properties of the services.

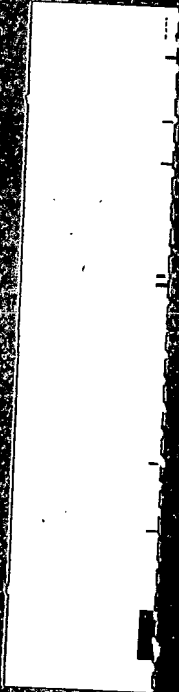
17. The system of claim 16, wherein a New Agent Request interpreter uses the list of previously used services with their interface definitions, keywords, descriptions, and related scenarios to automatically translate user requests into service interfaces and scenario actions.

18. The system of claim 17, wherein the New Agent Request interpreter uses a list of previously used services with their interfaces, keywords, descriptions, and related scenarios to offer selected parts of this information to the user for semi-automated translation of new requests into service interfaces and scenario actions.

1c Jim Zhuk

11191 East Ida Place

Englewood, CO 80112



7006 2150 0000 0591 3610



U.S. POSTAGE
PAID
ENGLEWOOD, CO
80112
MAR 12, 07
AMOUNT
\$5.36
00095331-10



22313

0000

RETURN RECEIPT
REQUESTED

Attention: Office of Petitions
Mail Stop Petitions
Commissioner for Patents
P.O. Box 1450
Alexandria, VA 22313-1450

RECEIVED

MAR 14 2007

USPTO MAIL CENTER



RECEIVED

MAR 15 2007

OFFICE OF PETITIONS